# Quantitative Safety: Linking Proof-Based Verification with Model Checking for Probabilistic Systems

Ukachukwu Ndukwu *
Department of Computing
Macquarie University, NSW 2109, Sydney, Australia.
ukndukwu@science.mq.edu.au

This paper presents a novel approach for augmenting proof-based verification with performance-style analysis of the kind employed in state-of-the-art model checking tools for probabilistic systems. Quantitative safety properties usually specified as probabilistic system invariants and modeled in proof-based environments are evaluated using bounded model checking techniques [4].

Our specific contributions include the statement of a theorem that is central to model checking safety properties of proof-based systems, the establishment of a procedure; and its full implementation in a prototype system (YAGA) which readily transforms a probabilistic model specified in a proof-based environment to its equivalent verifiable PRISM [10] model equipped with reward structures [1]. The reward structures capture the exact interpretation of the probabilistic invariants and can reveal succinct information about the model during experimental investigations. Finally, we demonstrate the novelty of the technique on a probabilistic library case study.

## 1 Introduction

There are two main techniques for investigating quantitative properties of *probabilistic* systems behaviours:

*Probabilistic model checking* comprises the formalisation of a model (which describes a system operationally), and a suite of algorithms to analyse various correctness and performance properties (usually expressed as a probabilistic temporal logic) of the model.

*Proof-based verification* on the other hand are practical applications of deductive proof methods to establish a link between the operational description of the model and the desired properties.

Over the years these two techniques have developed almost independently. The goal of this paper is to establish a formal link between them in a way that has never been previously explored. Our intention is to make such linkage beneficial to practitioners of both techniques, and hence ensure future applicability of the proposed practice especially on an industrial scale.

The B-Method [3] is an industrial-strength specification language for describing large-scale abstract system behaviours. The method's development process ensures that specifications gradually evolve via *refinement* to implementable code. The probabilistic B (pB) [7] extends the B-Method to incorporate probability.

PRISM [10] is a probabilistic model checker which accepts probabilistic models described in its modeling language — a simple state-based language. Three types of probabilistic models are supported directly — Discrete Time Markov Chains (DTMCs), Markov Decision Processes (MDPs), and Continuous Time Markov Chains (CTMCs). This work is based on the the MDP type of the language.

*The author is a recipient of the Australian Commonwealth Endeavour International Postgraduate Research Scholarship (E-IPRS) while pursuing a Ph.D. in the Information and Networked Systems Security (INSS) research group at Macquarie University, Australia.

S. Andova et.al. (Eds.): Workshop on Quantitative Formal Methods: Theory and Applications (QFM'09)
EPTCS 13, 2009, pp. 27–39, doi:10.4204/EPTCS.13.3

One of the key features of the pB language is the statement of invariants — they provide a means of describing properties required to maintain the integrity of a system under construction. In an attempt to establish a generalised view of probabilistic system invariant properties, Hoang T.S. *et al.* [14] originally explored the use of "expectations" as system invariants in proof-based systems. Including probabilistic invariants in systems design is a useful means of enforcing quantitative safety properties — for example, tolerance for expected error in a probabilistic system behaviour.

Discharging a pB machine's safety proof obligation involves the verification (by a human or automated support) that indeed the expectation holds for all executions of the machine; but sometimes the prover fails to establish this goal. It is pertinent to mention that not all the undischargeable proof obligations are malignant to the overall performance outlook of the final machine for deployment. However, in safety-critical systems development, this assumption cannot be taken lightly hence the need to explore other techniques of gaining intuition into the failure of the provers to discharge their proof obligations.

For standard (non-probabilistic) B machines, a prototype system [15] which incorporates a model checking tool has been used to detect various errors in simple machine specifications. However, for the probabilistic counterparts, we show how to link the model checking facility of PRISM with abstract systems specified in pB via the latter's probabilistic invariants. The importance of such a link is to enable pB designers explore their models experimentally; such an exploration is likely to reveal undesirable performance attributes of their models, and in particular guide them with a decision in the event that the invariants fail to hold.

Our technique is as follows. Given a proof-based machine specified in pB, we generate its equivalent probabilistic action systems representation (in the PRISM language) fully augmented with reward structures [1] inherited from the pB machine's expectations, and either confirm (or refute) the statement over the expectations. In the event that the statement fails to hold, we get an intuition of the possible cause(s) of failure. Our specific contributions are as follows:

(a) The statement of a theorem that forms the implicit link between a pB invariant and its reward-based model checking equivalent.

(b) A procedure for transforming a pB machine specified in a proof-based environment to its equivalent PRISM representation.

(c) A prototype system to automate the procedure. In addition, we equip the resultant PRISM model with reward structures inherited from the pB machine's expectations, to allow for experiments.

(d) A demonstration of the novelty of our technique on a small case study of a probabilistic library system.

Overall, this paper is structured as follows. We introduce the pGSL and its expectations in sec. 2, and set the theoretical foundation of our procedure in secs. 3, and 4. The automation of that procedure is in sec. 5. A practical demonstration of our technique is in secs. 6 and 7. Finally, we conclude in sec. 8.

## 2 Probabilistic Generalised Substitution Language *pGSL*

Abrial's Generalised Substitution Language *GSL* [3] is based on Dijkstra's weakest-precondition *wp* semantics of describing computations and their meaning [5]. The semantics, expressive in the *B-Method* (B) [3], defines the concept of an "abstract machine". The *Abstract Machine Notation (AMN)* explores B's capabilities via *refinement* for incrementing designs such that relevant system properties are always preserved. The complete framework supports the development of provably correct systems.

| command name | command (*comm*) | transformer semantics ($wp.comm.E$) |
|---|---|---|
| identity | skip | $E$ |
| assignment | $x := f$ | $E[x := f]$ |
| composition | $r; r'$ | $wp.r.(wp.r'.E)$ |
| choice | $r \lhd G \rhd r'$ | $wp.r.E \lhd G \rhd wp.r'.E$ |
| probability | $r \ {}_p\!\oplus\ r'$ | $wp.r.E \ {}_p\!\oplus\ wp.r'.E$ |
| nondeterminism | $r \sqcap r'$ | $wp.r.E$ min $wp.r'.E$ |
| strong iteration | do $G \to r$ od | $\mu X \bullet (wp.r.X \lhd G \rhd E)$ |
| weak iteration | It $r$ tI | $\nu X \bullet (wp.r.X$ min $E)$ |

Figure 1: Structural definition of the expectation transformer-style semantics.

The logic pGSL [11] is a smooth extension of the GSL, in which the standard boolean values — representing certainty are replaced by real-values — representing probability. Its logical framework is the *probabilistic Abstract Machine Notation (pAMN)* [7], an extension of the standard AMN. Its specification is based on the probabilistic B (pB). The syntactic structure of the pGSL is rich enough to permit the specification of abstract probabilistic system behaviours. Details of the GSL can be found in [3], while that of its theoretical and practical extensions are in [11] and [14, 7] respectively.

An important component of the pGSL is the specification of probabilistic invariants to ensure consistency between system designs. This is indeed crucial since it also assures that undesirable operation sequences do not lead to a violation of the critical properties of a system. In fact, the semantics of the pGSL which is based on the expectation transformer-style semantics of pGCL [9] (shown in Fig. 1) gives a complete characterisation of probabilistic programs with nondeterminism, and they are sufficient to express many performance-style properties. To further explore this capability, we set out the definitions below. Their elementary details can be found elsewhere [13, 16].

***Definition 1:*** *(Sub-distribution) For any finite state space S, the set of sub-distributions over S is*

$$\overline{S} \triangleq \quad \{\Delta : S \to [0,1] \mid \sum \Delta \leq 1\}\ , \tag{1}$$

*the set of functions from S into the closed interval of reals [0, 1] that sum to no more than one* [1].

***Definition 2:*** *(Labeled Markov Decision Process) A tuple (S, ŝ, A, L), where S is as defined above, $\hat{s} \in S$ is the initial state, $A : S \to 2^{\overline{S}}$ is a transition function, and $L : S \to 2^{AP}$ is a labeling function which assigns to each state, a subset of the set of atomic propositions AP that are valid for that state.*

***Definition 3:*** *(Path) A path in an MDP is a non-empty finite or infinite sequence of states $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1}$ ... where $\alpha_i \in A(s_i)$ and $\alpha_i(s_{i+1}) > 0$ for all $s_i \in S$.*

***Definition 4:*** *(Absorbing state) A state $s_i \in S$ is said to be absorbing if no transition leaves this state after resolving all the nondeterministic selections in the state i.e., $s_i \xrightarrow{\alpha_i=1} s_i$, and $s_i \xrightarrow{\alpha_i=0} s_j$ whenever $i \neq j$.*

A probabilistic computation tree formalises the notion of a probabilistic distribution over execution traces required to give semantic interpretation to temporal properties. Each step on a path will have an associated probability (often probability one for standard steps in the computation) — and the probabilities on those individual steps when multiplied together, determine probability for paths ending up in a particular absorbing state. Our interest lies in only using such probabilities masses for *finite* paths.

[1] Probabilities that sum to less than one represent aborting behaviours. We do not discuss such program behaviours here.

***Definition 5:*** *(Endpoint of a distribution) Any absorbing state s over the distribution $\Delta$ is said to be at the endpoint of the distribution.*
***Definition 6:*** *(Random variable) A random variable is a non-negative real-valued function over the state space in which our programs operate.*
***Definition 7:*** *(Expected value) For any bounded random variable $\alpha$ in $S \rightarrow \mathbb{R}_{\geq}$ and distribution $\Delta \in \overline{S}$, the expected value of $\alpha$ over $\Delta$ is defined*

$$\int_{\Delta} \alpha \triangleq \sum(\alpha.s * \Delta.s) , \tag{2}$$

*for any state s in the endpoint of the distribution $\Delta$.*

## 2.1 The PCHOICE Operator

In [14] Hoang T.S. *et al.* introduced a **PCHOICE** operator in the standard AMN's operations — similar to the probabilistic choice operator of Fig. 1, which also permits the specification of probabilistic behaviours in a typical machine. This extension, captured in the the probabilistic Abstract Machine Notation (pAMN), and expressed in the pB method, describes probabilistic machines with an additional EXPECTATIONS clause[2]. Ideally, probabilistic invariant properties are then defined as random variables over the machine's state, and encoded in the EXPECTATIONS clause. An invariant of this form is then an "expected value-invariant". Later on, we show how the pAMN can be used to specify abstract probabilistic system behaviours. A comprehensive list of the pAMN clauses can be found in [7].

## 2.2 The EXPECTATIONS clause

It gives a random variable $\xi$ over the program state, denoting the expected value-invariant, and an initial expression $e$ which is evaluated over the program variables when the machine is initialised. The idea is that after arbitrary executions of the program, the expected value of $\xi$ at any given program state, is always at least the value of $e$ initially [14].

More formally, suppose a probabilistic machine has initialisation *INIT* and two operations *OpX* and *OpY* respectively, therefore, satisfying the probabilistic proof obligation for some expected value-invariant $\xi$ and initial expression $e$ would operationally (Fig. 1) imply that[3]

$$\xi \Rrightarrow wp.OpX.\xi \qquad \text{and} \qquad \xi \Rrightarrow wp.OpY.\xi \quad \text{provided} \quad e \Rrightarrow wp.INIT.\xi, \tag{3}$$

which then assures that

$$e \Rrightarrow wp.INIT.(wP.(\text{It } OpX \sqcap OpY \text{ tI}).\xi). \tag{4}$$

The operational interpretation of (4) is that arbitrary interleaving of the operations *OpX* and *OpY* after the initialisation *INIT* should always result in a distribution over the final states (of variables) such that the expected value (with respect to invariant $\xi$) is at least the initial value specified by the expression $e$. Clearly, the conditions in (3) imply the operational interpretation of (4). However, if there is a particular interleaving of the machine which demonstrates the failure of (4), then it must be true that (3) has hitherto failed as well. The example below illustrates our argument.

[2]The complete framework encapsulates state variables and the operations on the states by the use of 'clauses'.
[3]For random variables $R$, $R'$, the implication-like relation $R \Rrightarrow R'$ means $R$ is everywhere less than or equal to $R'$.

**MACHINE** Demon
**SEES** Int_TYPE, Real_TYPE
**VARIABLES** $cc$
**INVARIANT** $cc : INT$
**EXPECTATIONS** $real(0) \Rrightarrow cc$
**INITIALISATION** $cc := 0$
**OPERATIONS**
$nn \leftarrow$ OpX = **BEGIN**
**PCHOICE** $frac(1,2)$ **OF** $cc := cc+1$
**OR** $cc := cc-1$ **END** $||$ $nn := cc$
OpY = $cc := 0$
**END**

Figure 2: A pB model specified in the pAMN.

### 2.2.1 Example: A Simple Demonic Machine

Fig. 2 shows a pAMN (adapted from [14]) that captures the operations of a simple pB machine called Demon, with a single variable *cc*; the INVARIANT clause specifies that *cc* must be Integer-valued — pB's prover always checks that this statement is true using the operational reasoning established in the previous section. Initially, *cc* is set to 0; the OPERATIONS clause contains operations *OpX* and *OpY*. *OpX* can either increment *cc* by 1 or decrement it by the same value both with probability $1/2$, while *OpY* just re-initialises *cc* to 0. The variable *nn* in the operation *OpX* is an output parameter which need not occur in the VARIABLES clause[4]. The EXPECTATIONS clause specifies the expected value-invariant $\xi$ to be the random variable *cc*, and the initial expression *e* to be 0, so that "the expected value of *cc* over any endpoint distribution is never decreased below 0" by the Demon's *OpX* and *OpY* operations.

In this example, the machine *fails* to satisfy the probabilistic proof obligation specified in (4), and the reason is that

$$(\exists cc \in INT : \ \neg(cc \ \Rrightarrow \ wp.OpY.cc)). \tag{5}$$

The immediate expression captures the failure of the pB prover to establish the proof obligations in (3). However, in terms of a distribution viewpoint, it is possible to see exactly why this failure of the pB prover would similarly result in the failure of (4). Consider the program fragment

$$INIT; OpX; (OpY \lhd (nn \geq 0) \rhd \text{skip}); \tag{6}$$

it yields the distribution given by

$$\delta = \begin{cases} cc := 0 & @1/2 \\ cc := -1 & @1/2 \end{cases}$$

over the final state of the random variable *cc*. Calculating the expected value over this distribution we get

$$\int_{\delta} cc = 1/2 \times 0 + 1/2 \times -1 \quad \equiv -1/2,$$

which is clearly a violation of the conditions specified in the EXPECTATIONS clause.

In this simple case, it is clear that the failure to establish the pB proof obligation corresponds to an exact result distribution over endpoints that demonstrates the failure. Currently pB provers do not

[4]It must however follow a similar machine declaration as *cc* to enable its PRISM transformation.

provide diagnostic information necessary to give practitioners the much needed operational intuition (in terms of distributions) for locating failures. This becomes even more complicated with increasing size and complexity of the pB machines. For such cases, we simply rely on the model checking capabilities of state-of-the-art tools like PRISM. To do this, we need to translate the pB machines to their equivalent PRISM models, and use the latter's algorithmic analysis to attempt to locate failures.

In the sections that follow, we show how the analysis of an equivalent PRISM representation of a pB machine can provide a link between the operational viewpoints (distribution-centered) of both a proof-based environment (encapsulating probabilistic invariants), and a model checking platform. Such a link is key to getting a better understanding of the expected value-invariants over endpoint probabilistic distributions.

## 3 PRISM Reward Specification

The PRISM model checker permits models to be augmented with information about rewards (or costs). The tool can analyse properties which relate to the expected values of the rewards. A reward structure [1] can be used to represent additional information about the system the MDP (the model types in this paper) represents — for example, the expected number of packets sent (or lost) on a protocols request.

The temporal logic probabilistic CTL [12] has been extended in [2] to allow for reward-based specifications as constraints of the type that express *reachability*, *cumulative* and *instantaneous* rewards to model checkers. However, for the purpose of this work, the instantaneous variant is useful.

***Definition 8:*** *(Expected instantaneous reward) The probabilistic CTL permits reward properties of the form* $R_{\sim r}[\mathbb{I}^{=k}]$*, at time-step k, where* $\sim \in \{<, \leq, \geq, >\}$*,* $r \in \mathbb{R}_{\geq}$ *and* $k \in \mathbb{N}$*. The reward formula* $R_{\sim r}[\mathbb{I}^{=k}]$[5] *is true if from some initial state* $s_0$*, the expected state reward at time-step k meets the bound* $\sim r$*. For example the specification* $R_{\geq 50}[\mathbb{I}^{=2}]$ *could be interpreted to mean that the expected number of packets sent by the protocol after two time-steps is at least 50.*

## 4 Quantitative Safety and pB Machines

Formal approaches necessitate that every safe system be *invariant-driven* [5]. Therefore, quantitative safety properties can be proved by verifying invariants. The EXPECTATIONS clause of a pB machine encapsulates the machine's safety property. However, an interesting dimension in investigating pB safety properties is whether it is possible to find a nondeterministic selection $\sqcap_{0 \leq n \leq N} P_n$ of the operations of the machine that demonstrates the failure of the invariants — this schedule must then lead to the problem of locating counterexamples for probabilistic model checking [6].

In [8] McIver *et al.* set out a strategy for computing a "refutation-of-safety" certificate for probabilistic systems using model checking techniques. The existence of a certificate corresponds to an invariant failure in our own context of safety. In the next section, we present an automation which demonstrates the key features of that strategy and in particular show how it can be used to investigate pB safety properties. The theorem below is fundamental for investigating safety properties specified for pB machines.

***Theorem:*** *Given a pB machine invariant* $\xi$ *and initial expression e encapsulated in the machine's EXPECTATIONS clause, let* $\Delta$ *be the finite result distribution over endpoints for the interleaving* It $\sqcap_{0 \leq n \leq N}\ P_n$ tI *after the machine's initialisation INIT. A safe machine always guarantees that*

$$e \Rrightarrow wp.INIT.(wp.(\text{It } \sqcap_{0 \leq n \leq N}\ P_n \text{ tI}).\xi) \Rightarrow \int_{\Delta} \xi \geq e, \tag{7}$$

[5] For MDP's we require Rmin or Rmax; this is allowed in PRISM by enabling the sparse engine in the tool's options menu.

*provided the pB machine's prover will always discharge the proof obligations given by* $e \Rrightarrow wp.INIT.\xi$ *and* $\xi \Rrightarrow wp.(\text{It} \sqcap_{0\leq n\leq N} P_n \text{ tI}).\xi$ respectively.

***Corollary:*** *Suppose it is always possible to split the distribution* $\Delta$ *into finite sub-distributions* $\delta_k$ *for all* $k \geq 0$ *where* $\delta_k$ *is the result distribution on the kth iteration, then*

$$(\forall k \geq 0 \wedge \delta_k \in \Delta : \quad \int_{\delta_k} \xi \ \geq \ e) \ . \tag{8}$$

The usefulness of the corollary is in the practical demonstration of the theorem. Its simplicity is captured by the fact that, if a probabilistic computation tree which interprets the execution traces of the machine is available to a verifier, then these traces must be finite with respect to the result distributions they represent. Using bounded model checking techniques, it then suffices to argue that only finite values of $k$ are required to establish the proof obligation for a safe pB machine. More so, with state-of-the-art probabilistic model checking tools such as PRISM, it is possible to identify a sub-space of the entire distribution in which the failure is located (if any).

# 5 YAGA: A pAMN To PRISM Translator

In section 4, we stated a theorem that is central to defining safety features for pB machines with a finite trace distribution over endpoints. The corollary of that theorem has the practical interpretation that, bounded model checking techniques when equipped with reward structures are likely to provide an intuition to locating invariant failures in their transformed proof-based models. In this section we discuss a language-level translator nicknamed `YAGA`[6] and with architectural framework shown in Fig. 3. `YAGA`, a java-based implementation of the algorithm pAMN2PRISM (shown in Appendix A) is a prototype system that essentially takes a pAMN framework, encapsulating a pB model (with syntactic checks supposedly discharged) as its input parameter, and generates its precise probabilistic action systems representation in the PRISM language. The associated reward structure of the generated PRISM model is inherited from the pB machine's EXPECTATIONS clause. The PRISM model checker then readily offers its temporal logic specification (as probabilistic CTL formulas) which can easily be checked by conducting experiments on the transformed model. The experimental results are sufficient to validate (or refute) the probabilistic invariants specified in the abstract machine's EXPECTATIONS clause.

## 5.1 Overview: YAGA Transformation Rules

We summarise the transformation rules as follows. YAGA's algorithmic interpretation is in Appendix A.

### 5.1.1 Main Module

**PRISM constants list:** Are constructed from the pB machine's parameter list (if any) and the CONSTANTS clause. The type of a constant is implicitly checked from the PROPERTIES clause.
**PRISM formula list:** Are auto-generated as *atomic* predicates from the pB machine's PROPERTIES and INVARIANTS clauses.
**PRISM module name:** Is the pB machine's name.
**PRISM variables declaration and initial values list:** Are constructed from each variable in the VARIABLES clause, its type in the INVARIANT clause, and its initial values from the INITIALISATION

[6]The name YAGA is coined from an Igbo (a language largely spoken in southeastern Nigeria) word — YAGAzie, which literally means "may it go well ...". In reality, it could be argued that YAGA is simply Yet Another Gangling Automation.

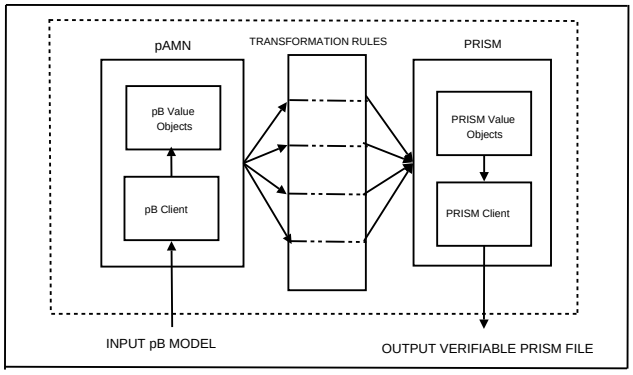


Figure 3: YAGA - Architectural Overview

clause. The lower and upper limits of the variables are respectively the default lowest values of their types, and a bound specified from the PRISM constants list (above).

**PRISM update statements:** Each update statement is labeled with the operation's name from the pB machine. In addition,

(a) its guard is inherited from the guard in the pB machine's OPERATIONS clause and strengthened by the formulas in the PRISM formula list, such that

(b) the choice of a selection of formula is dependent on the expressions in the pB machine's update statement. For each update, YAGA checks that the formula-dependent expressions are included in the PRISM guard.

### 5.1.2 Counter Module

The counter module is a vital encoding that helps enumerate the distributions in $\delta_k$ (in corollary) for finite $k$ steps. To capture this behaviour in a model checking environment, we apply the following rules.

**PRISM module name:** Counter.

**PRISM variable declaration and initial value:** Variable *count* is initially set to 0 and bounded by $(MAX\_COUNT+1)$.

**PRISM update statement:** Each update is constructed to synchronise with the updates in the main module. They can only increment the *count* variable by 1 on each action. In addition, this module also contains a similar unsynchronised update statement which ensures we will eventually reach $(MAX\_COUNT+1)$.

### 5.1.3 Reward Structure

The specific reward structure is inherited from the pB machine's EXPECTATIONS clause — states where the *count* variable equals $(MAX\_COUNT+1)$ are worth the random variable value specified in the EXPECTATIONS clause plus $(MAX\_COUNT)$[7].

However, to make the construction of our reward structures precise for model checking, we note more formally as a result of the theorem in sec. 4 that

***Remark 1:*** *Given any pB machine invariant $\xi$ and initial expression $e$, then from an initial state $s_0$, after the machine's initialisation INIT, any arbitrary interleaving* It $\sqcap_{0\leq n\leq N}\ P_n$ tI *must guarantee that:*

$$(k :\in [0, MAX\_COUNT+1] : e \Rrightarrow wp.INIT.(wp.(\text{It } \sqcap_{0\leq n\leq N} P_n \text{ tI}).\xi) \Rightarrow Rmin_{=?}[\mathbb{I}^{=k}] \geq \xi.s_0) \qquad (9)$$

[7]This padding is to ensure the PRISM engine is consistent with computing positive instantaneous rewards. Finally, we subtract this parameter from the PRISM computed reward value.

*such that the expected minimum instantaneous reward at the kth step is worth the random variable value of the EXPECTATIONS clause plus MAX_COUNT. We recall that the Counter module keeps explicit track of the kth time-step, and the expected value here is captured by the sub-distribution in (8).*
***Remark 2:*** *However, if there exists some k such that (9) above fails to hold, then $\xi$ cannot be an invariant.*

# 6 Case Study: A Library Bookkeeping System

We present a pB machine which captures the basic operations underlying the accounting package of a library system — the implication of an undischargeable proof obligation of the machine on the performance of the library was an open problem in [14]. The state of the machine contains four variables: *booksInLibrary*, *loansStarted*, *loansEnded* and *booksLost* which are respectively used to keep track of: the number of books in the library, the number of book loans initiated by the library, the number of book loans completed by the library, and the number of books possibly never returned to the library.

Initially, the machine has two operations: *StartLoan*, to initiate a loan on a book, and *EndLoan*, to terminate the loan of a book. The *StartLoan* operation has a precondition that there are books available for loan; it decrements *booksInLibrary* and increments *loansStarted*; when a book is returned, the *EndLoan* operation reverses the effect of the *StartLoan* operation by recording that either the book "really is" returned, or is actually reported lost with some probability $pp$, so that *booksLost* is incremented.

The machine uses the random variables *loansEnded* and *booksLost* to record the expected losses of the number of books over time. Since with a probability $pp$ a book is lost on each *EndLoan* operation, the library system would then be expected to lose a proportion $pp$ over a number of *EndLoan* operations. However, to ensure that the library is always in the business of books lending, we define the expected value-invariant

$$\int_{\Delta} (pp \times loansEnded - booksLost) \geq 0, \tag{10}$$

which captures the idea that the expected value of the random variable $pp \times loansEnded - booksLost$ over its endpoint distributions $\Delta$ can never be decreased below 0. This is indeed a safety property for the library and ought to be checked throughout its lifetime to ensure it is not violated by its future designs. Below, we present two designs of the library and also keep in mind the property specified in (10).

## 6.1 A Safe Library Bookkeeping System

Since there are no restrictions on when the operations of the machine can be invoked, except for the obvious preconditions on *StartLoan* and *EndLoan*; the specification of a safe library system is the non-deterministic choice given by

$$SafeLibrary \triangleq \text{It } StartLoan \sqcap Endloan \text{ tI} . \tag{11}$$

## 6.2 An Unsafe Library Bookkeeping System

Suppose that to enable the library accountant do a periodic stock take of the library transactions, a new operation called *StockTake* is introduced into the system. The *StockTake* operation is very similar to the initialisation, except for an extra output (*totalCost*) to record the cost of replacing the books lost up to the time of doing a stock take. Again, we augment (11) and give another specification of the library as

$$UnsafeLibrary \triangleq \text{It } StartLoan \sqcap Endloan \sqcap StockTake \text{ tI} . \tag{12}$$

The complete pB machine describing the library (all of its three operations) is shown in Appendix A.

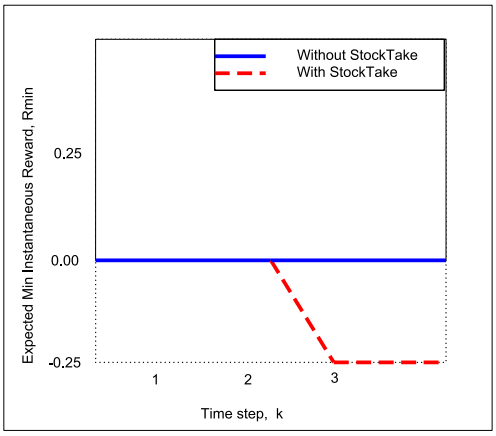


Figure 4: Library Bookkeeping System

## 7 PRISM Experimental Results

In this section we report experimental results that are indeed performance-style characterisations of the two designs of our library model — the safe library (11) and the unsafe library (12). Our interest lies in justifying the reason why one design of the library (without stockTake) is safe and why the other (with stockTake) is unsafe. We note that our safety property of concern is captured by (10).

To enable us carry out this performance analysis, we quickly use the capabilities of YAGA. The equivalent PRISM representation of the pB machine discussed in the previous section as generated by YAGA is shown in Appendix A. From Remark 1, our obvious reward specification becomes

$$(\forall\, 0 \leq k \leq MAX\_COUNT : \quad Rmin_{=?}[\mathbb{I}^{=k}] - MAX\_COUNT) \,. \tag{13}$$

The requirement for a safe library system is that for all time-steps $k$, *Rmin* is never decrease below zero. However, our experimental result (shown in Fig. 4) reveals that indeed the unsafe library violates this safety property after just three time-steps ($k$ = 3) of its execution — that is for $MAX\_COUNT$ = 2, $pp = 0.5$, and $totalBooks = 1$, the expected minimum instantaneous reward *Rmin* of the unsafe library is -0.25.

A quick conclusion that can be drawn from our analysis is that introducing the "demonic" StockTake operation has the adverse effect of subverting the overall performance outlook of the library system. Our result is a practical demonstration of the claim by Hoang T.S. *et al.* [14] in an attempt to explain why the presence of the StockTake operation would result in a failure of the proof obligation for the probabilistic invariant property in (10). In the proof-based system, reaching this conclusion was practically impossible.

## 8 Conclusion and Future Work

This paper has explored the practical application of reward-based specifications of bounded model checking techniques [4] to locate failures in the context of proof-based verification for simple safety properties of probabilistic systems. We demonstrated the rich benefits that can be derived by complementing proof-based probabilistic verification techniques with a model checking performance-style evaluation, in a manner that has never been previously explored.

Our contribution is seen as a first attempt at fully integrating quantitative performance analysis to systems design at early stages of development. Our method scales in this regard since it can be carried

out at the level of source code and hence can guide system developers with decisions on a choice of design most suitable for implementation.

However, in other to fully integrate this performance-style analysis into software development process, we intend to, in the future, incorporate a diagnostic mechanism based on counterexamples location employed in [6, 17] to YAGA. Our intention is that such a mechanism will report explicit cause(s) of failure by also exploring the backward analysis strategy in [8]. It is our belief that these enhancements would provide a useful performance analysis suite for probabilistic systems developed in the pB language.

## 9 Acknowledgement

The author is grateful to his thesis supervisor, A.K. McIver for her unflinching research guidance. We also wish to express our gratitude to Dave Parker of the Oxford University computing laboratory for his technical input on matters relating to the PRISM tool. Finally, to the anonymous reviewers whose comments have been invaluable in improving the preliminary draft of this paper, we say a big thank you.

## References

[1] Kwiatkowska M., Norman G. and Parker D. (2007): *Stochastic Model Checking*. *In Proceedings of SFM'07* vol. 4486 LNCS, 220-270. Springer, 2007.

[2] Andova S., Hermanns H., Katoen J.-P. (2003): *Discrete-Time Rewards Model-Checked*. *In Proceedings of Formal Modeling for Timed Systems (FORMATS'03)* vol. 2791 LNCS, 88–104, Marseille, France, 2003.

[3] Abrial J. R. (1996): *The B Book. Assigning Programs to Meanings.* Cambridge University Press, 1996.

[4] Biere A., Cimatti A., Clark E.M., Strichman O. and Zhu Y. (2003): *Bounded Model Checking*. *Advances in Computers*, vol. 58, 2003.

[5] Dijkstra E.W. (1976): *A Discipline of Programming*. Prentice Hall International, Englewood Cliffs, N.J., 1976.

[6] Han T. and Katoen J.-P. (2007): *Counterexamples in Probabilistic Model Checking*. *In Proceedings of TACAS*, vol. 4420 LNCS, 72-86. Springer, 2007.

[7] Hoang T.S. (2005): *Developing a Probabilistic B-Method and a Supporting Toolkit*. PhD Thesis, UNSW.

[8] McIver A.K., Morgan C.C. and Gonzalia C. (2008): *Probabilistic Affirmation and Refutation: Case Studies*. *In Proceedings of APV 2009. URL:* `http://se.ethz.ch/apv/program.html`

[9] McIver A.K. and Morgan C.C. (2004): *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer Verlag, 2004.

[10] PRISM: *Probabilistic Symbolic Model Checker*. URL: `http://www.prismmodelchecker.org/`

[11] Morgan C.C. (1998): *The Generalised Substitution Language Extended to Probabilistic Programs*. *In Proceedings of B'98: The 2nd International B Conference*, vol. 1393 LNCS, Montpelier, April 1998.

[12] Hansson H. and Jonsson B. (1994): *A Logic for Reasoning about Time and Reliability*. *Formal Aspects of Computing*, 6(5):512-535, 1994.

[13] Grimmett G.R. and Welsh D. (1986): *Probability: An Introduction*. Oxford Science Publications, 1986.

[14] Hoang T.S., Jin Z., Robinson K., McIver A.K. and Morgan C.C. (2003): *Probabilistic Invariants for Probabilistic Machines*. *In Proceedings of ZB'03*, vol. 2651 LNCS, 240-259. Springer Verlag, 2003.

[15] Leuschel M. and Butler M. (2003): *ProB: A model checker for B. In proceedings of FME'03 vol. 2805 LNCS, 855-874. Springer Heidelberg, 2003.*

[16] Clarke E.M., Grumberg O. and Peled D. (1999): *Model Checking*. MIT Press, 1999.

[17] Aljazzar H. and Leue S. (2007): *Counterexamples for Model Checking of Markov Decision Processes. Computer Science Technical Report soft-08-01, University of Konstanz, December 2007.*

## Appendix A

**MACHINE** ProbabilisticLibrary (totalBooks, cost)
**SEES** Real_type
**CONSTANTS** pp
**PROPERTIES** $pp \in REAL \wedge pp \leq real\ (1) \wedge real\ (0) \leq pp$
**VARIABLES** *booksInLibrary, loansStarted, loansEnded, booksLost, totalCost*
**INVARIANT** *booksInLibrray, loansStarted, loansEnded, booksLost, totalCost* $\in$ *NATURAL* $\wedge$ *loansEnded* $\leq$ *loansStarted*
$\wedge$ *booksInLibrary + booksLost + loansStarted - loansEnded = totalBooks*
**EXPECTATIONS** $real\ (0) \Rrightarrow pp \times real\ (loansEnded) - real(booksLost)$
**INITIALISATION** *booksInLibrary, loansStarted, loansEnded, booksLost, totalCost := totalBooks, 0, 0, 0, 0*
**OPERATIONS** StartLoan = **PRE** *booksInLibrary* $>$ *0* **THEN**
*booksInLibrary := booksInLibrary - 1* || *loansStarted := loansStarted + 1* **END**;
EndLoan = **PRE** *loansEnded* $<$ *loansStarted* **THEN**
**PCHOICE** *pp* **OF** *booksLost := booksLost + 1*
**OR** *booksInLibrary := booksInLibrary + 1* **END** || *loansEnded : = loansEnded + 1* **END**;
StockTake = **BEGIN** *totalCost := cost* $\times$ *booksLost* || *booksInLibrary := booksInLibrary + booksLost* ||
*loansStarted := loansStarted - loansEnded* || *loansEnded := 0* || *booksLost := 0* **END**
**END**

Figure 5: The pB Model of Section (6)

**const** *totalBooks;*
**const** *cost;*
**const double** *pp;*
**const** *MAX_COUNT;*

**formula** *formula0* = *(loansEnded* $\leq$ *loansStarted);*
**formula** *formula1* = *(booksInLibrary + booksLost + loansStarted - loansEnded = totalBooks);*
**formula** *formula2* = *(pp* $\leq$ *1);*
**formula** *formula3* = *(0* $\leq$ *pp);*

**module** ProbabilisticLibrary
*booksLost:[0..totalBooks]* **init** 0;
*totalCost:[0..totalBooks]* **init** 0;
*loansEnded:[0..totalBooks]* **init** 0;
*loansStarted:[0..totalBooks]* **init** 0;
*booksInLibrary:[0..totalBooks]* **init** *totalBooks*;

*[StockTake] formula1 & formula0* $\rightarrow$ *(totalCost′ = cost * booksLost) & (booksInLibrary′ = booksInLibrary + booksLost)*
*& (loansStarted′ = loansStarted - loansEnded) & (loansEnded′ = 0) & (booksLost′ = 0);*
*[StartLoan] (booksInLibrary* $>$ *0) & formula1 & formula0 & (loansStarted + 1* $\leq$ *totalBooks)* $\rightarrow$
*(booksInLibrary′ = booksInLibrary - 1) & (loansStarted′ = loansStarted + 1);*
*[EndLoan] (loansEnded* $<$ *loansStarted) & formula2 & formula1 & formula3 & formula0* $\rightarrow$
*pp: (booksLost′ = booksLost + 1) & (loansEnded′ = loansEnded + 1)*
*+ (1 - pp): (booksInLibrary′ = booksInLibrary + 1) & (loansEnded′ = loansEnded + 1);*
**endmodule**

**label** *"expectations" = (pp * loansEnded - booksLost* $\geq$ *0);*

**module** Counter
*count:[0..MAX_COUNT + 1]* **init** 0;

*[StockTake] (count + 1* $\leq$ *MAX_COUNT + 1)* $\rightarrow$ *(count′ = count + 1);*
*[StartLoan] (count + 1* $\leq$ *MAX_COUNT + 1)* $\rightarrow$ *(count′ = count + 1);*
*[EndLoan] (count + 1* $\leq$ *MAX_COUNT + 1)* $\rightarrow$ *(count′ = count + 1);*
*[] (count + 1* $\leq$ *MAX_COUNT + 1)* $\rightarrow$ *(count′ = count + 1);*
**endmodule**

**rewards**
*(count = MAX_COUNT + 1) : (pp * loansEnded - booksLost) + MAX_COUNT;*
**endrewards**

Figure 6: A YAGA-Generated PRISM Representation of Fig. (5)

Algorithm pAMN2PRISM (pB_model_in_pAMN)

Required: An interface for pB syntax, PRISM syntax, and regular math operators.
Reserved: MAX_COUNT (constant), count (variable)

1: get pAMN parameter list if any
2: create a map object with pAMN clauses as the the keys. Insert their respective values
3: construct value objects with (1) and (2)
4: set module type as MDP
5: construct PRISM constants list from the pAMN parameter list and PROPERTIES key
6: construct PRISM formula list as atomic predicates from the INVARIANT and PROPERTIES keys
7: get PRISM module name from MACHINE key (declare module name)
8: construct PRISM variables list and their initial values from the VARIABLES and INVARIANT keys and the pAMN parameter list (if any)
9: **for** the OPERATIONS key in the map object **do**
  get the list of operations
  **for** each operation in the list **do**
    **for** each guard and update statement in operation **do**
      check variables dependency on (6)
10: construct PRISM update statements with the pAMN operations names as the action labels
11: declare **endmodule**
12: construct expectations label from the EXPECTATIONS key in map object
13: declare **module** counter
14: declare count variable, initialised to zero and bounded by MAX_COUNT + 1
15: **for** the OPERATIONS key in the map object **do**
  get the list of operations
  **for** each operation in the list **do**
    construct a synchronised update statement (increment) on the count variable with (10)
16: construct an unsynchronised update statement on the count variable
17: declare **endmodule**
18: declare PRISM **rewards**
19: for states where (count = MAX_COUNT + 1)
20: set reward to random variable value on EXPECTATIONS key plus MAX_COUNT
21: declare **endrewards**

Figure 7: An Algorithmic Description of YAGA